Mapod 设计文档

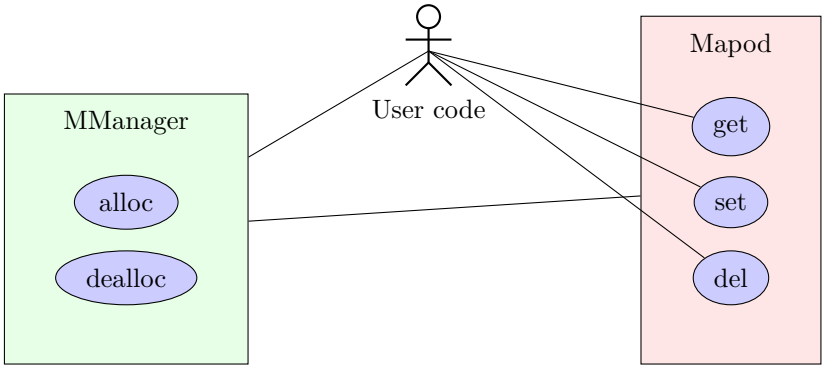
钱泽森 (5130379069)

January 2, 2015

Contents

1 模块设计

1.1 用例图



1.2 功能设计说明

1.2.1 `Mapod<Key, T>`

Mapod 负责 B+ 树的插入/删除/查找, 存储部分的操作则通过 `MManager` 来完成.

1.2.2 `MManager`

MManager 主要负责文件空间 (包括索引文件和数据文件) 的分配与释放, 文件到内存的映射. 简单起见, 目前采用的分配方式是 append-only, 不回收释放的磁盘空间.

这部分接口也可以被用户代码使用, 供用户实现自定义的序列化函数, 请见样例程序.

2 样例程序

2.1 简单样例 (`demo.cpp`)

一个简单的样例, 实现 `<long, long>` 的 Key-Value 存储

```
#include "mapod.hpp"
#include <iostream>
using namespace std;

int main()
```

```
{
  Mapod<long, long> map("map");
  string command;
  long key, t;
  while (cin >> command >> key) {
    if (command == "set") {
      if (cin >> t)
        map.set(key, t);
      else
        cerr << "invalid value" << endl;
    } else if (command == "get") {
      cout << map.get(key) << endl;
    } else if (command == "del") {
      map.del(key);
    } else {
      cout << "Unknown command" << endl;
    }
  }
}
```

2.2 使用自定义序列化 (demo1.cpp)

默认的序列化/反序列化函数只实现了 shallow dump, 对于部分数据通过指针引用的情况无法处理. 因此我提供了接口, 供用户使用自己的序列化/反序列化函数, 来实现 deep dump.

```
#include "mapod.hpp"
#include <iostream>
#include <string>
using namespace std;

off_t dump(const string &t, MManager &mm)
{
  const off_t off = mm.alloc(t.size() + 1);
  char *ptr = (char *)mm[off];
  strcpy(ptr, t.c_str());
  return off;
}

string load(const off_t off, MManager &mm)
```

```
{
  const char *ptr = (char *)mm[off];
  return string(ptr);
}

void destroy(const off_t off, MManager &mm)
{
  mm.dealloc(off);
}

int main()
{
  //remove("map.node");
  //remove("map.data");
  DLD<string> dld{dump, load, destroy};
  Mapod<string, string> map("map", Mapod<string, string>::compare, dld, dld);
  string command;
  string key, t;
  while (cin >> command >> key) {
    if (command == "set") {
      if (cin >> t)
        map.set(key, t);
      else
        cerr << "invalid value" << endl;
    } else if (command == "get") {
      cout << map.get(key) << endl;
    } else if (command == "del") {
      map.del(key);
    } else {
      cout << "Unknown command" << endl;
    }
  }
}
```

3 接口设计

3.1 Mapod<Key, T>

- 构造

```
Mapod(const std::string &prefix, const KeyCompare *compare = compare,
      const DLD<Key> &key = DLD<Key>::defaultDLD,
      const DLD<T> &t = DLD<T>::defaultDLD,
      const int k = 0, const off_t blksize = 512);
```

  - prefix: 数据文件的前缀
  - compare: 比较函数, 在 k0 严格小于 k1 时返回 true
  - key: 一系列序列化/反序列化函数, 负责 Key 的 dump/load/destroy
  - t: 同上, 但是负责 T 的 dump/load/destroy
  - k: 分支系数 (branching factor), 给定 0 则会根据 blksize 自动计算最佳系数
  - blksize: 底层块设备的块大小 (在机械硬盘上一般为 512 bytes)

- 插入

```
void set(const Key &key, const T &t);
```

  - key: 待插入的键
  - t: 待查入的值

- 查找

```
T get(const Key &key);
```

  - key: 欲查找的键
  - @return: 查找到的值. 如果没有找到则返回 T()

- 删除

```
void del(const Key &key);
```

  - key: 欲删除的键

3.2 MManager

- 申请空间

```
off_t MManager::alloc(const off_t size, const off_t align = 4);
```

  - size: 申请的空间大小

  - `align`: 空间对齐要求
  - `@return`: 分配的空间在文件中的位置

- 释放空间

```
void MManager::dealloc(const off_t off);
```

  - `off`: 释放空间在文件中的位置

- 重新分配空间

```
off_t MManager::realloc(const off_t off, const off_t
size, const off_t align = 4);
```

  - `off`: 待调整的空间原位置
  - `size`: 申请的新的空间大小
  - `align`: 空间对齐要求
  - `@return`: 新分配的空间在文件中的位置

4 测试

4.1 正确性测试 (test.cpp)

正确性测试采用如下步骤:

1. 选用 STL 的 `std::map` 作为对照
2. 依次测试最大 key 数 2,4,8,16, … , 1048576 的情况
3. 对于每种情况, 分别测试 1,2,3,4,…,20 次
4. 在每次测试中, 都进行 (key 数)^3 次操作. 操作可能是插入或是删除, 随

机决定. 每次操作后, 都会随机查找一个 `key` , 检查返回的 `t` 与 `std::map` 的返回值是否相同.

对 `Mapod<long, long>` 和 `Mapod<string, string>` 均进行以上测试. 测试中没有发现错误.

4.2 性能测试 (`benchmark.cpp`)

CPU	Intel(R) Core(TM) i3-4010U CPU @ 1.70GHz
CPU cache	3072 KB
Keys	8 bytes each
Values	8 bytes each
Entries	1048576
Raw Size	16 MB (estimated)

性能测试采用如下步骤:

1. 获取当前 key 数
2. 插入随机的 2^{13}(8192) 对 KV, 并记录这批插入操作所花的时间.
3. 查找随机的 2^{13}(8192) 对 KV, 并记录这批查找操作所花时间.
4. 插入随机的 2^{13}(8192) 对 KV, 并记录这批插入的所有 key.
5. 根据 3 的记录, 删除第 3 步插入的 key, 并记录这批删除操作所花时间.

重复上述步骤 2^7(128) 次, 直到总键值数到达 2^{20}(1048576) 次为止. 这样我们就得到了 128 组数据, 分别反映了单次的插入/查找/删除操作在数据库大小在 $[0, 2^{20}]$ 之间的耗时.

4.2.1 默认配置

默认配置下, Mapod<long, long> 的性能测试结果如图1所示. 可以看到, 单次操作耗时与 key 数基本成对数级关系. 即使在 4 百万条数据的情况下, 插入一条数据仍然只需要大约 1000ns(1μ s), 相当于 1M/s 的吞吐量. 另外还可以观察到, 删除和查找相对较快, 而插入则相对较慢些.

4.2.2 Inline Vs. External

如果单个 `Key` 的大小实际上不大于 `off_t` (用来记录实际数据块在数据文件中的偏移), 那么再把这个 `Key` 存到数据文件里就不划算了. 默认的 `Dump/Load` 会检测 `Key` 以及 `T` 的大小, 如果数据类型不大于 `off_t` , 那么该数据会直接嵌 (inline) 在索引文件中, 不涉及数据文件. 同样对于 `<long, long>` 的维护, 对 inline 和 external 的存储模式分别进行测试, 结果如图2所示. 可以看到, inline 的性能差不多是 external 的两倍. 有趣的是 external 初期 (在 0M~0.4M 之间) 的剧烈抖动, 我至今仍然不知道是怎么回事, 只能推测是由于同时高频读写两个文件 (索引文件和数据文件), OS 的 cache 机制在这种情况下表现不佳.

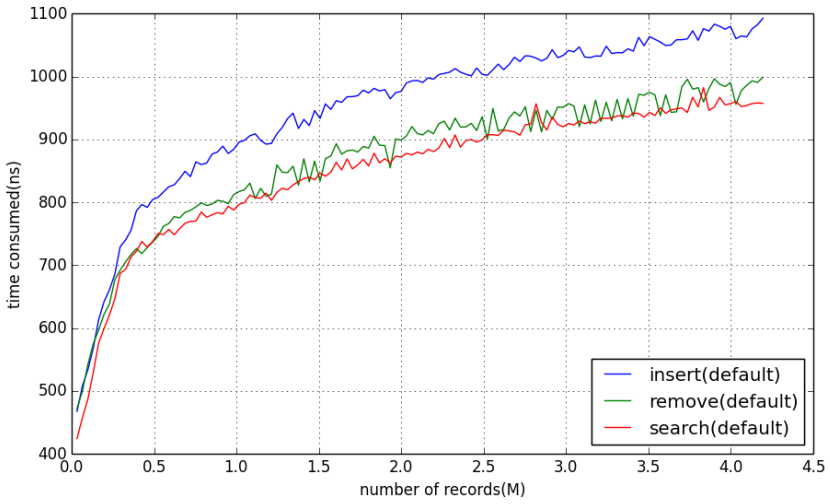


Figure 1: 默认配置下 <long, long> 单次操作耗时

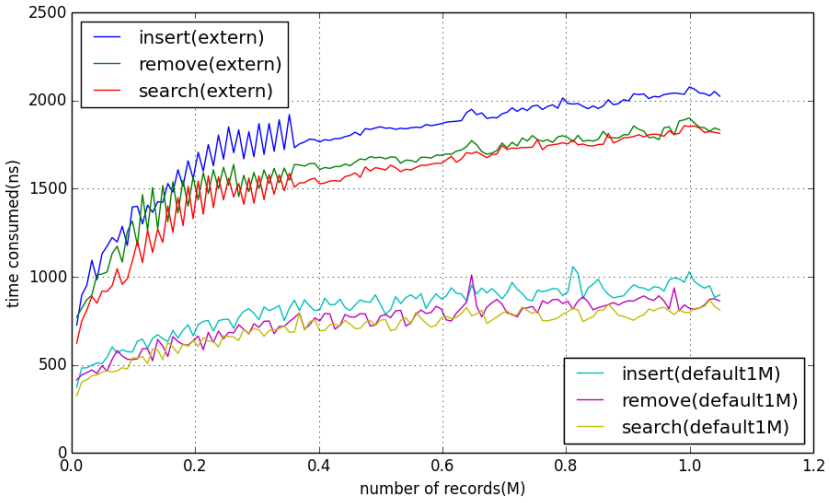


Figure 2: Inline 和 External 的存储方式对比

4.2.3 The optimal `blksize`

调整不同的 `blksize` , 性能测试结果如图3所示. 可以观察到, 对于查找操作, blksize 为 1024 时最快, 512 次之, 256 最慢. 推测是由于更大的 blksize 带来更大的 `k` (branching factor), 所以访问的节点个数更少, 速度更快. 对于插入和删除操作, 可以看到 blksize 为 256 时最快. 这是由于插入和删除均需要在节点内做大量的移动, 所以较小的 blksize 移动的内容较少, 速度较快. 注意由于测试的数据不是很大, OS 把我们的所有索引都缓存在了内存中, 所以本测试与块设备 (磁盘) 的块大小没有直接的明显的关系.

4.2.4 Lazy or Not Lazy

如果将 `benchmark.cpp` 中的删除语句 `dmap.del(inserted[j]);` 换成插入语句 `dmap.set(inserted[j], 0);` , 那么我们就实现了 lazy deletion, 并且我们的 benchmark 程序正好保证了数据库里有一半" 已经被删除, 但仍然存在" 的元素. 测试结果如图 4所示.

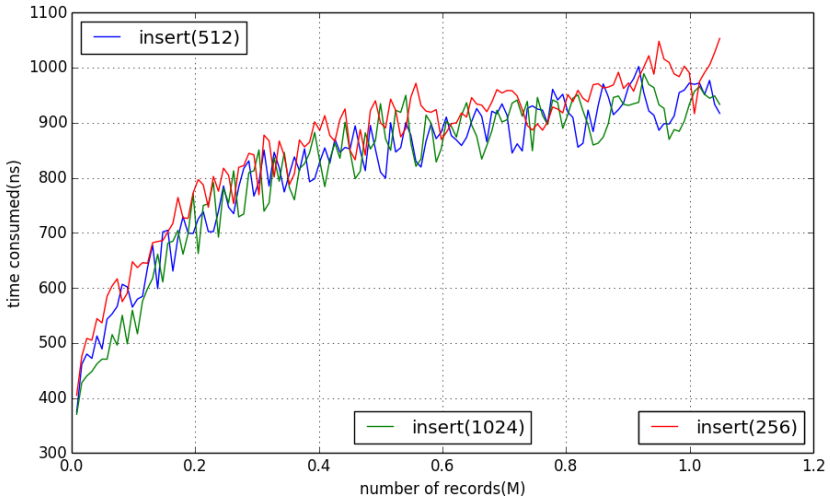
设想中, 由于 lazy deletion 不需要在删除时维护树的结构, 所以应该比普通的 deletion 快一点. 但是测试结果却表明 lazy deletion 的三种操作均明显慢于普通实现. 对此我也无法解释.

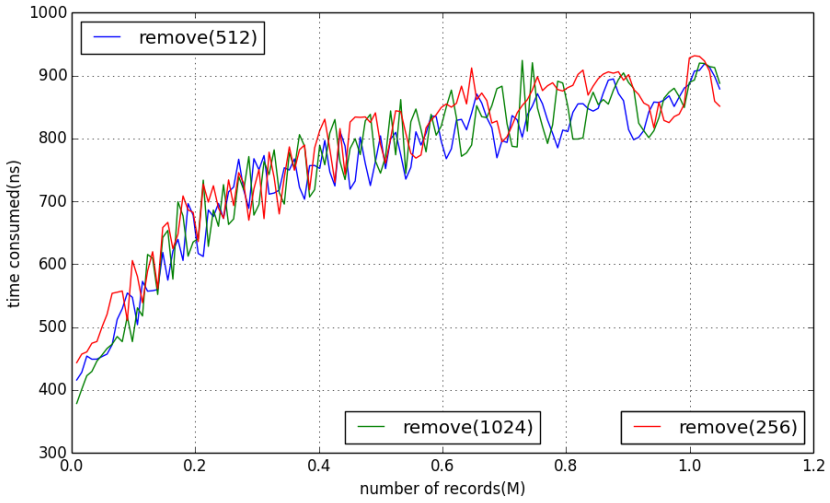
4.2.5 OS cache

对更大的数据量进行测试 (4M+), 测试结果如图5所示. 从图中可以看到, 在数据量小于 5M 时, 单次操作的耗时几乎没有变化, 曲线几乎是水平的. 这也反映了我们的算法是对数级的.

但是在大于 5M 后, 曲线出现了剧烈的波动. 单次操作的耗时从 1 μ s 级剧烈到达了 20 μ s 级别, 而这个数量级恰好与测试机的机械硬盘的平均寻道时间吻合. 因此这个现象可以解读为, 在 key 数大于 5M 后, OS 的 cache 机制决定不再把我们的所有数据都 cache 在内存中, 此时访问部分节点会导致访问硬盘, 因此带来了波动. 经计算 5M 的 <long, long> 实际占用的空间大约在 80Mb 左右, 远远小于本机空闲内存 (G 级别). 这也说明了 OS 的 cache 是通用的 cache 机制, 不适合数据库占用较大空间的情况. 解决改问题的方式有如下几种:

- 调优内核的参数, 使得 OS 倾向于使用更多的 cache.
- 使用 `mlock()` 函数, 提示 OS 将我们的数据锁定在内存中, 不要 swap 到硬盘. 但是 `mlock()` 只是对 OS 的提示, 在内存不足的情况下 OS 仍然不会对我们的数据 cache.
- 实现自己的 cache 机制. 自行决定将哪些数据载入内存, 自行管理一致性. 这个较复杂, 所以我没有实现.





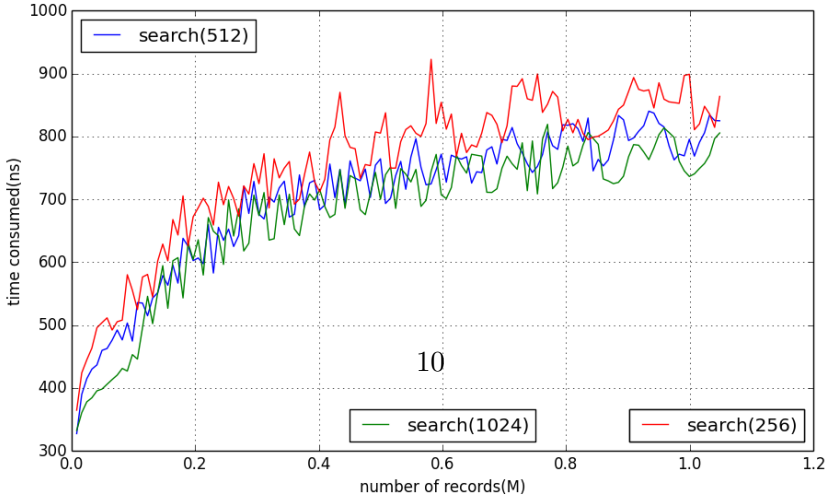


Figure 3: 不同 blksize 下的性能对比

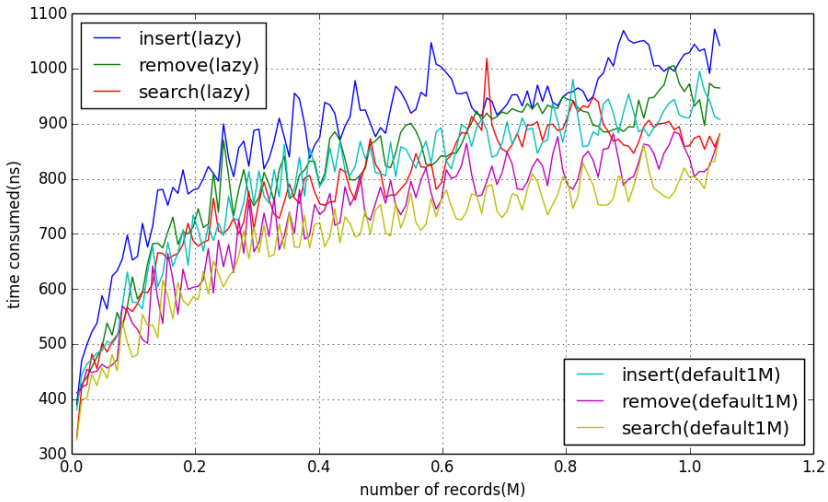


Figure 4: lazy deletion 与默认实现对比

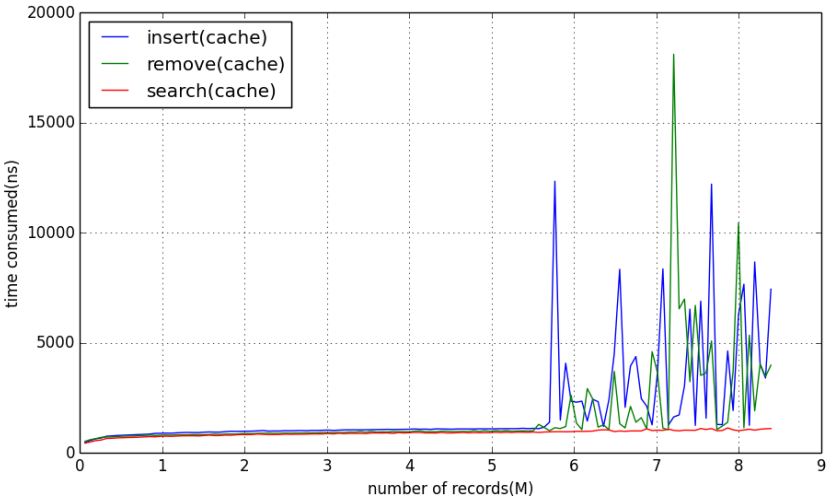


Figure 5: 大数据下 OS cache 失效